Copyright © 1982 American Telephone and Telegraph Company
THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 61, No. 9, November 1982
Printed in U.S.A.

Database Systems:

A Directed Hypergraph Database: A Model for the Local Loop Telephone Plant

A. J. GOLDSTEIN

(Manuscript received November 5, 1981)

A database model that is a directed hypergraph is described in the context of an application to the telephone plant. There is particular emphasis on the fidelity of the application model to the "real world," information hiding by modules, transaction concurrency, names of inventoried objects, and pending. Pending refers to the maintenance of a tree of possible future states of the inventory.

I. INTRODUCTION

The hypergraph database model was designed and implemented to support an application that deals with the topology of communication circuits, not their transmission properties. The application program inventories, administers, assigns, and assembles the communication circuits connecting the switching entities in the central office to a customer's premises. (See Fig. 1.) (For historical reasons we will use the term "living unit".) The word "loop" in the title refers to such a circuit, while "local" specifies that the loop does not use more than one switching entity.

A number of difficult problems arise in modeling the local loop telephone plant. I hope to show that the hypergraph database provides a basis for conceptually clean solutions and that it encourages modularity and information hiding. There are no controlled experiments in a very large project so there is the pain of seeing the large warts in the way "you did it," but no pleasure in knowing how much larger the warts would be if "you had done it the other way".

The database described here is a bare bones, do-it-yourself, specially designed version of an entity-relationship database.[1] While most of the

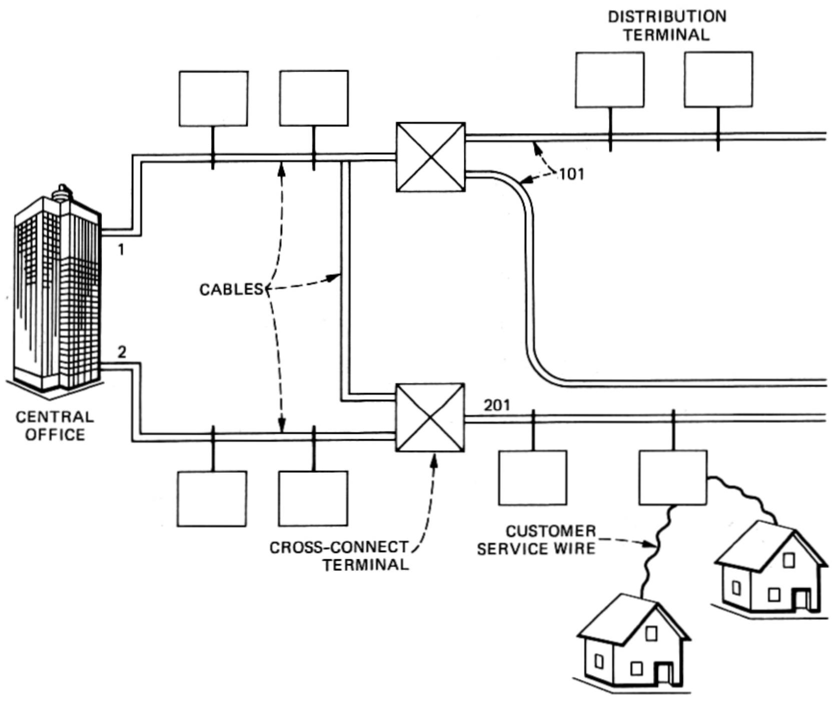

Fig. 1—Portion of local loop plant.

topics transcend the loop plant application, the discussions use the specific loop in Fig. 2 as a canonical example.

The objects inventoried are:

(*i*) Pair—a pair of wires.

(*ii*) Cable—an administrative or physical set of pairs.

(*iii*) Terminal—a device in which the ends of pairs are connected to the beginnings of others. Each connection is made at a terminal's binding post.

(*iv*) Binding posts—devices in a terminal to which wires are connected.

(*v*) Living unit—a house, apartment, office, suite, pier, slip, wharf, ··· .

(*vi*) Customer service wire—the pair that connects the living unit to a nearby terminal.

(*vii*) Loop—the collection of objects required to provide a communication path from the customer's living unit to the central office: living unit, customer service wire, and one or more binding posts, terminals, and pairs.

Figure 1 shows a portion of a local loop plant. Some pairs in cable 1 are connected to binding posts on the IN side of the upper cross-

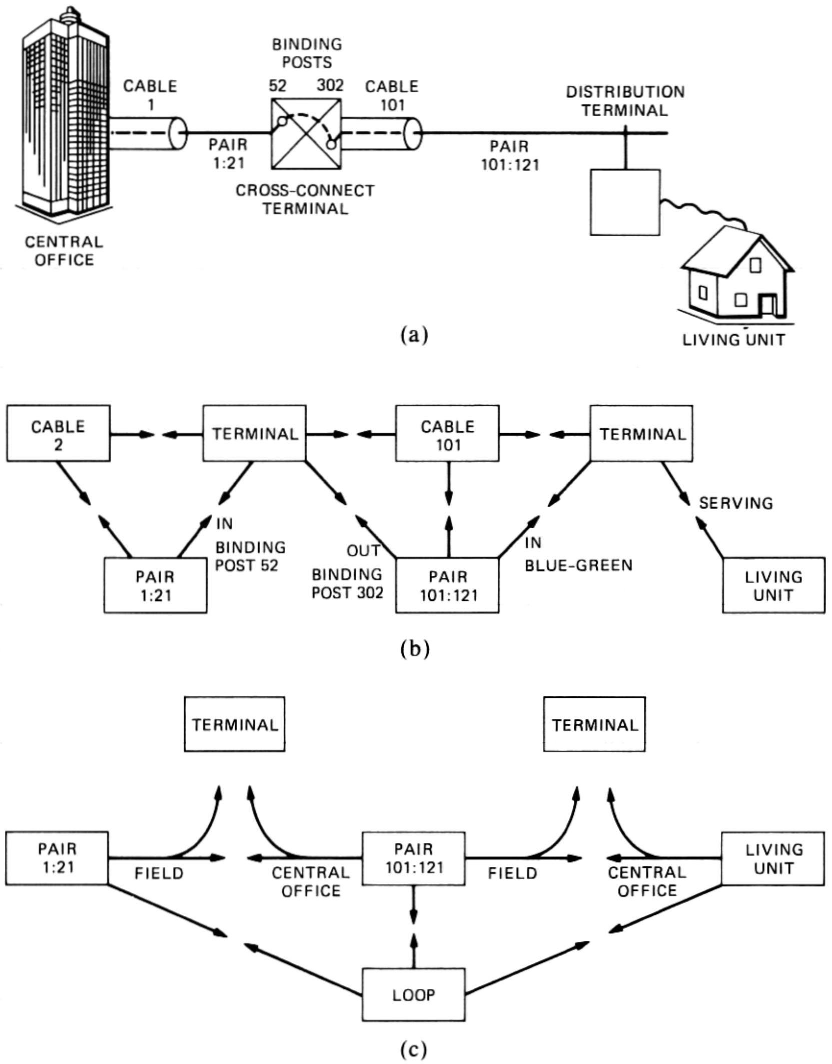

Fig. 2—The canonical example. (a) The real world. (b) Inventory. (c) Connectivity.

connect terminal, other pairs are connected to binding posts on the IN side of the lower cross-connect terminal. Pairs in cable 101 are attached to binding posts on the OUT side of the upper terminal. An IN pair from cable 1 and an OUT pair from 101 can be interconnected (called cross connected) by electrically connecting the corresponding IN and OUT binding posts. Similarly, a pair from cable 1 or from cable 2 can be cross connected to one in cable 201 in the lower cross-connect terminal.

Since, as the example shows, a pair may appear in many terminals, the topology of the loop plant is not simple. The application is complex because of the topology and the repeated modification of the loop plant by additions, deletions, and rearrangements of cables and their pairs.

Our paper consists of a series of sections that successively elaborate the information included. We provide below an outline of the paper as an aid to the reader.

1.1 Terminology

(*i*) An *object* refers to a real-world physical or administrative entity.

(*ii*) A *node* is a database representation of an abstraction of both the attributes of an object and its relationships to other objects.

(*iii*) The *external name* of an object is the name used by people to identify the object. The external name is an attribute of a node.

(*iv*) The terms program, routine, function, and logic are used interchangeably. *Application programs* (routines, functions, or logic) are programs that implement the loop plant requirements.

(*v*) Each node in the database has two formats. One, the *program record* format, is the structure of the node as declared in the syntax of the native programming language. In our case, the language is *C* and the node's program record is a *C* structure.[2] The other format, the *file record* format, is the structure of the node in the syntax of the file system. The difference between the formats is explained in the section on the General Record Interfacing Technique (GRIT).

(*vi*) Each node is given at birth a unique identifier, its *file-id*. A node's file-id never changes. It is independent of its external name, which may change. Nodes refer (i.e., point) to each other by their file-ids. To track the present and (possibly multiple) future states of an object, each of its states is represented by a node. Each such node has, of course, its own file-id but also contains the datum, *base-id*, which is the file-id of the node representing the present state. (See Sections IV on pending.)

1.2 Caveat emptor

There are numerous simplifications of the facts; they are sometimes errors of omission to avoid details that are irrelevant or too messy; they are sometimes errors of commission to simplify the presentation of a point. None distort the essentials.

II. THE MODEL OF THE LOCAL LOOP PLANT

The project began by developing an abstract model of the loop plant. The modeling concentrated on (*i*) abstracting the relevant physical and administrative objects and their attributes, and (*ii*) abstracting the relationships among the objects. For example, a pair has the attributes:

status:	working or idle
restriction:	yes, no
remarks:	insulation starting to fray
defect:	yes, no, why (low-impedance short, 60-cycle hum, . . .)

Some of a pair's relationships are to:

cable:	element of
terminal:	appears in
	side (in, out)
	binding post number
	color
pair & terminal:	connected to the pair in the terminal in the direction of (central office or field)

Figure 2a illustrates a loop that we will use as a canonical example throughout the paper. Some of the objects and their relationships are illustrated in Figure 2b where each square represents the object above it. The relationship of pair 21 to the cross-connect terminal is represented by the directed edge containing the data showing that the pair is on the IN side of the terminal and is on binding post 52. The relationship between pair 121 and the terminal is similar. The directed edges between each cable and pair represent "element of" and "contains" relationships. Pair 121 appears on the IN side of the distribution terminal and inside the terminal it can be identified by blue-green insulation. (A pair is sometimes spliced outside of a terminal to a stub that is already attached to a binding post. The pair may be spliced to a different-colored stub at each terminal.) The living unit is "served by" the terminal and the terminal serves the living unit. To simplify the example, the customer service wire is not shown. The terminal-to-cable edges represent redundant information that is contained in the path from cable to pair to terminal.

Figure 2c gives the connectivity relationships of the communication circuit. The loop contains two pairs and a living unit. The two-headed edge at pair 1:21 shows that the pair is connected to pair 101:121 in the terminal pointed to. The other edges give analogous relationships. We shall return to these edges in the next section.

III. THE HYPERGRAPH DATABASE MODEL

The database model is an abstraction of the loop mode. It is a generalization of a directed graph; data on the vertices represent attributes of physical and administrative objects; data on the directed edges represent attributes of relationships. Since, as discussed below, simple directed edges were insufficient to model the complex relationships of the loop plant. The graph was generalized to a hypergraph.[3] Recall that a directed graph is a structure with a set of vertices and a set of directed edges each of which originates at one vertex and terminates at one other vertex. A directed hypergraph extends this notion by replacing "one" by "one or more." One can think of the directed hyperedge as a fork with one or more tines in which the handle originates at one vertex and each tine terminates at another.

The following terminology is used:

(*i*) *Body* denotes a vertex of the hypergraph and its data.

(*ii*) *Edge* denotes a directed hyperedge and its data

(*iii*) *Node* denotes a body plus all the edges that originate at the body.

Each node represents an object. The body data are the object attributes. The edge data are the attributes of the relationships of the object to the object(s) to which the edge points. Figure 3 shows pair 121 in cable 101 and its relationships. Figure 4 is an annotated example of the contents of that pair node. Edges are not ordered in the present implementation, and to find a particular edge, one must search for it. This figure (with some minor expository simplifications) is exactly what the node's file record looks like. The file record is a string of characters (including newlines and tabs), and if printed, would look like that figure. GRIT (discussed below) converts the node's file record to a program record and vice versa.

The database model was first developed as a directed graph. The complexity of connectivity relationships was the impetus for evolving to a hypergraph model. For example, in Fig. 2c we see that each hyperedge represents two facts: (*i*) a pair is cross connected to another

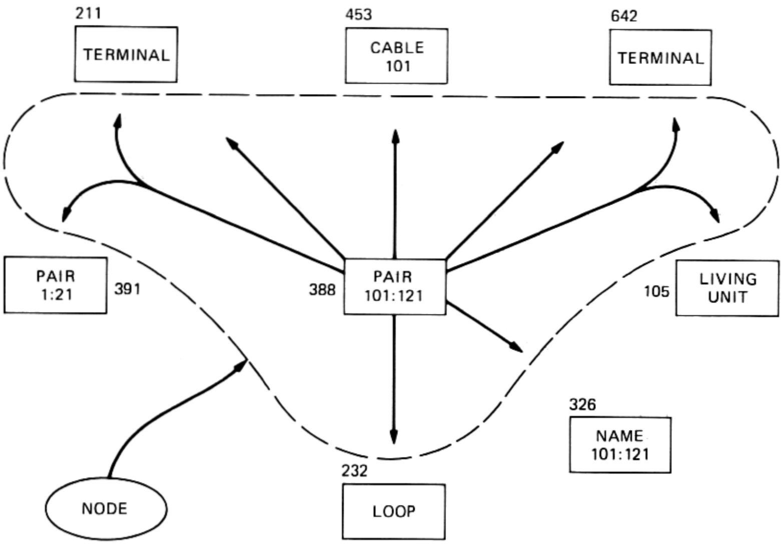

Fig. 3—Node for pair 121 in cable 101.

pair (or living unit) and (*ii*) the pairs are cross connected in a particular terminal. What is to be represented is not merely that the pair is in one (or several) terminals nor that the pair is connected to one (or several) other pairs, but this pair is connected to that pair in that terminal.

To illustrate the complexity of the cross-connect relationship in a directed graph database model, let us examine two alternative representations in such a database. One uses logic to discover the relationship, while the other keeps the relationship in a relationship record.

In the first alternative there are directed edges among each of two cross-connected pairs and their terminal. (There are six edges among the three objects.) Then, given one pair, a cross-connected partner is found by searching the pair's node for an edge to a pair. A search of the pairs' edges for a common terminal identifies the one in which they are cross connected. Each time a similar relationship arises, special logic would have to be created.

In the other alternative, a relationship record is created to which the two pairs and terminal point. This alternative comes to grips with the relationship by forcing a database model onto the problem. It requires that every relationship be implemented by the artifice of a special record. The addition of a relationship will always require a reorganization of the database.

Note that the hypergraph model has the relationship in a pair's hyperedge, while the first alternative requires the pair's partner pair

```
body                                  data about the pair itself
        type      pair                        type of this node
        id        388                         self identification
        rmk       f2 pair                     second from central office
        f1        no                          not at central office

edge                                  pair is element of
        type1     cable
        id1       453
        rltn      deletor                     if delete cable (deletor),
                                              then delete pair (deletee)

edge                                  pair is cross connected
        type1     pair                        to
        id1       391
        type2     terminal                    in
        id2       211
        path      cent. off.                  toward

edge                                  pair appears
        type1     terminal                    in
        id1       211
        bndpst    302                         on binding post
        side      out

edge                                  pair appears
        type1     terminal                    in
        id1       642
        color     blue-green                  with the color
        side      in

edge                                  pair connected
        type1     terminal                    in
        id1       642
        type2     lu                          to
        id2       105
        path      field                       toward

edge                                  name node
        type1     name
        id1       326
        exid      0101:121                    cable 0101 pair 121

edge                                  pair is in
        type1     loop
        id1       232
```

Fig. 4—Contents of node for pair 121 in cable 101.

to be read and the second alternative requires the relationship record to be read.

The hypergraph database is a very special entity-relationship database.[1] Relationships are restricted to those that can be represented by directed hyperedges. Entities and relationships (node bodies and hyperedges) are as structures (in the sense of *C* or *PL/I*). No compile time checks of entities and relationships are made other than those for *C*, the implementation language.[2] The development of our own hypergraph database management system (HDBMS) is the price we paid for using the hypergraph database instead of a conventional one. See Section IX on the database model for more details.

IV. PENDING

This application must reflect possible future states of the inventory. For example, if a customer requests that service be supplied at a living unit at a future date (the due date), then the database must reflect the state of the loop plant before the due date and the pending state after the due date. The programs that select facilities to satisfy the request use the before state of the inventory. The before state may be pending! In general, the pending states form a tree. A pending state becomes an actual state after the execution of a transaction informing the system that the service was installed. The cancellation of a customer's order causes the corresponding pending state and its successors to be deleted. Moreover, the customer requests that produced the successor states must be reprocessed.

Pending has been carefully designed to operate in ignorance of the application to the local loop plant and vice versa. This independence is crucial to the success of the project and is accomplished by an inversion of the usual program design.

A conventional approach to pending would have each application routine be aware of pending and be designed to handle all the various pending (and pending on pending) cases. This approach would couple application routines since a change in the way any one modifies pending states might affect many other routines. This direct approach was inverted so that, in some sense, pending drives the application logic and each application program is written as if pending did not exist. (This is a small lie since some loop plant requirements specify the pending status of objects.) See Section X on pending for more details.

V. CONCURRENCY

Performance requirements demand that, in a large local loop plant, transactions be able to process concurrently. Fortunately, analysis shows that a concurrency of 2 or 3 suffices. As will be seen, the relationships among nodes is too complicated for a page- or node-locking mechanism to work since deadlock would occur too frequently. Hierarchical locking is not possible because the topology of the loop plant is neither simple enough nor constant in time. In this application, a subgraph of the database hypergraph can be abstracted which can be used to implement a *dynamic hierarchical locking* that meets the low-concurrency requirement. The locking subgraph is a directed acyclic graph that reflects both the changing loop plant topology and pending. See Section XI on concurrency for more details.

VI. EXTERNAL NAMES AND INTERNAL IDENTIFIERS

There is a capricious world of names by which humans identify objects in the loop plant. Items can be classified as nice or nasty. There is one flavor of nice but many flavors of nasty.

- Some objects are nice; they have unique names (e.g., pairs).
- Some objects are unpleasant; they have aliases (e.g., two addresses for a living unit on a corner lot).
- Some distinct objects are confusing; they have the same name. (For example, a customer is moving to a new location and wants the same telephone number at the old and new locations during a transition period. Since, the telephone number is the external name of a loop, the loops at both locations have the same name.)
- Some objects are anonymous; they have no name (e.g., an idle loop—one with no customer assigned to it).
- Some objects are deliberately difficult; they have pending name changes (e.g., a telephone number change to obtain an unlisted number).

To insulate the database from these vagaries three steps are taken. First, every node, when created, is given an identifier, called the file-id. That file-id is the only way that other nodes' edges may refer to the node. The file-id never changes even if the external name does. Second, there is a table that maps the external name to the file-id of either the node representing the object referred to, or to a *name node*. The table is hashed on the external name. For nodes with nicely behaved external names (e.g., pairs, cables, or terminals) the file-id in the table is that of the unique node referenced by the external name. Third, for nodes whose external names are not nicely behaved (e.g., loops and living units) the internal identifier in the table is that of a name node. The name node's edges point to every node with the same external name. Similarly, every nasty node with aliases points to many name nodes. Name nodes can pend! The name node removes external names from the special case category and frees them to become first-class citizens with the same rights and privileges as any other item in the database.

This idea has converted some requirements from very difficult program design problems to easy ones.

Since external names can change erratically, the file-id serves the function of isolating such changes to three places: the external-name to file-id table, the node itself (it contains the external name in an edge), and the name node. Edges pointing to a node whose name changes do not have to be modified!

Less exception logic is required if the nice as well as nasty nodes have name nodes. But, it is too costly since about 70 percent of the nodes are nice. However, we assume, for ease of exposition, that every node with an external name has a name node.

The section on primitives shows how the name node is made transparent to the application routines.

VII. SYSTEM COMPONENTS

The system components are: the file system, the Hypergraph Database Management System (HDBMS), the pending routines, the node primitives (abstract data types), the loop plant application routines, the user of the system, and the database audit routines. Figure 5 in conjunction with the following sections will show the relationships among these modules. Recall from the terminology section that (*i*) a node's file record contains the data as stored in the file system, and

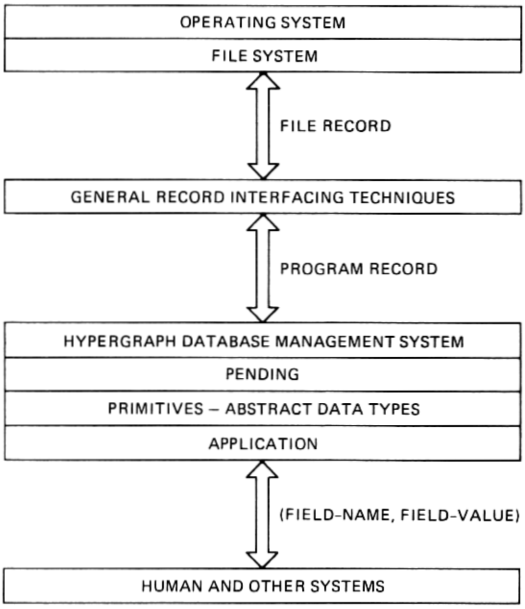
OPERATING SYSTEM
FILE SYSTEM
FILE RECORD
GENERAL RECORD INTERFACING TECHNIQUES
PROGRAM RECORD
HYPERGRAPH DATABASE MANAGEMENT SYSTEM
PENDING
PRIMITIVES – ABSTRACT DATA TYPES
APPLICATION
(FIELD-NAME, FIELD-VALUE)
HUMAN AND OTHER SYSTEMS

Fig. 5—System components.

(*ii*) the node's program record contains the data as stored in the structure defined by a declaration in the program's language syntax.

7.1 The file system

The file system stores one file record for each node. Each record is a string of characters whose structure is unknown to the file system. Since a node consists of a body and a set of edges whose number changes with time, its file record is implemented as a linked list of fixed length blocks. The location of the first block of characters in the record is calculated from the node's file-id. The file system knows almost nothing else about the database. "Almost" means it also stores the external-name to file-id table and does part of the locking for concurrency of transactions.

7.2 The Hypergraph Database Management System (HDBMS)

The HDBMS interfaces with the file system via a narrow window through which a node's file record (a character sequence) is fetched or stored. Each node's file record is converted into a program record (a body and edges) based on a format determined by the "node's type" (i.e., pair, cable, terminal, ...). An inverse conversion stores into the file system. The GRIT routines perform the conversions. The HDBMS routines also:

(*i*) Perform the basic functions on nodes: copy_next_edge to a buffer, and {add or delete or replace} a {node or body or edge}.

(*ii*) Maintain the external-name to file-id tables; the table's inversion is accomplished by having each node keep its own external name(s).

(*iii*) Manage the allocation and freeing of nodes and file-ids.

HDBMS is ignorant of the file system (almost), the hypergraphs, and the loop plant.

7.3 The general record interfacing technique (GRIT)

Often, when program changes occur in a large system, the database has to be reorganized merely to reflect a new data format in a record. In this way the size of the database (as opposed to its logical complexity) quickly becomes the limiting factor in the evolution of the system.

GRIT counters this limiting effect. It has a format document from which are generated two equivalent (but distinct) data representations: one defines the *file record* format in the file system and the other defines the *program record* format in the program. (The latter is in the native programming language's syntax.) Two I/O transfer functions convert from one format to the other.

The function of GRIT can be illustrated as follows. Suppose that program P-A reads and writes nodes using program and file formats

PF-1 and FF-1. Suppose also, that the data content of nodes must change, but only for another program, P-B. P-B must now read and write nodes using PF-2 and FF-2. As long as the two formats are related by a small set of simple rules, file records can be read using either P-A or P-B!

When say, P-A writes a file record, R-A (using file format FF-A) and P-B reads R-A (using program format PF-B) differences are reconciled by the following (recall all fields are character strings):

(*i*) Ignore a field in R-A that is not required by PF-B.

(*ii*) Use a null string for a field that is required in PF-B that is not in R-A.

(*iii*) Truncate or null fill a field in R-A that is required by PF-B, but whose length is wrong.

GRIT is of no help when reorganizations are due to the addition of a new type of record, the redistribution of data among records, the change in physical record size, or the change in physical record location.

7.4 Primitives—abstract data types

The primitives serve the function of being abstract data types; they hide the details of the node structure from the application programs. Thus, the node's structure and data content can change and only those parts of the application logic that need the new data are affected. The primitives have eased the pain of much of our development work.

For example, there is a set of primitives for edges of the form

fetch_next_edge_with_attribute(attr).

Also, there is the primitive

nbr = fetch_file_id(name, fid).

that, when given an external name, returns fid, the file-id of an object with that name, and nbr, the number of objects with that name. Fetch file id obtains the file-id corresponding to the name from the external-name to file-id table. However, that file-id is the file-id of the name node of the object! The name node is read, its first edge is examined, and the file-id of the node pointed to is returned in fid. The number of edges is placed in nbr. A subsequent call with the previous fid as the second argument causes fid to be replaced with the file-id of the next edge and nbr to be decremented.

7.5 The pending routines

The pending routines create new pending versions of nodes and keep track of pending relationships among pending versions. They hide from the application the fact that it is dealing with a pending version. Most application routines are designed as if there were no pending. However, application routines that must satisfy requirements specifically involving pending can obtain the pending status of nodes. The pending routines automatically update the inventory when a pending state becomes the actual state. They know nothing about the loop application and while the present implementation explicitly uses the hypergraph database, other implementations are possible.

7.6 The application (loop plant) routines

These routines know about the loop plant model and the nodes and their hyperedges. They communicate with the outside world of humans and machines by sets of field-name and field-value pairs that are character strings. They are ignorant of the HDBMS and (for the most part) of pending.

7.7 The database audit routines

The audit routines examine each node to verify that the attributes in its body and the relationships in its edges are correct. Application-dependent tables drive the audit routines, which are largely application independent. The table defines a schema that contains the valid node attributes and relationships. The schema on the hypergraph data base is not supported by the programming language's data structure syntax. Thus, there is no compile time validation of the hypergraph relationships. The table fills the gap caused by the syntax deficiency and the audit routines fill the gap caused by the deficiency in the compile validation.

For each node type (e.g., pair, cable, terminal, ...) the table defines:

(*i*) The attributes of the node's body and the set of permissible values of each attribute.

(*ii*) Possible relations of the node's body attributes to edge attributes.

(*iii*) For each edge type (i.e., type of node(s) pointed to):

- a. the number of such edges,
- b. the attributes of the edge and the set of permissible values of each attribute,
- c. the relationship between the attributes of the edge and those of the node(s) pointed to.

VIII. CHANGE TOLERANCE AND PORTABILITY

Change tolerance and portability are governed by the way the system is divided into components. The success of our partition can be assessed by a history that has traversed three operating environments, two computers, two major program designs, and one major enhancement. In Table I we see that the initial modeling design was done on a PDP 11/70 using the *UNIX* operating system (which is not a transaction environment) and the *C* programming language.[2] This model was ported to the UNIVAC 1100/83. The experience with the model led to a prototype that incorporated (at least in skeletal form) all the functions and data structures needed to reflect the loop plant requirements. The prototype ran in both the *UNIX* system and in the transaction (called execution) environments. Programs were developed in the very comfortable *UNIX* system environment. The prototype was expanded to meet all the loop plant requirements and then was installed at the user's site.

An important contribution to the change tolerance and portability are narrow windows to the file system and outside world. The file system knows only the fact that a node record is a variable-length string of characters. The outside world interacts via pairs of character strings (field-name, field-value).

IX. DATABASE MODEL—SUPPLEMENT

This section details the generic structure of a node. Figure 4 is the structure of one example of a pair, while Fig. 6 is the generic structure of all nodes.

9.1 File-identifier

See the external names and file identifiers section (Section VI).

9.2 Body

The body contains data for identification (node-type and base-id), data for locking, and data associated with the application.

Table I—Project history

Operating System	Computer	Date	Event
		2-79	MODEL
*UNIX**	PDP 11/70	8-79	
|	|		
UNIX	UNIVAC 1100/83	10-79	MODEL
			|
UNIX	UNIVAC 1100/83	5-80	PROTOTYPE
|			
EXECUT. ENV.	UNIVAC 1100/83	5-80	PROTOTYPE
			|
EXECUT. ENV.	UNIVAC 1100/83	11-81	DELIVERED TO USER

* *UNIX* is a trademark of Bell Laboratories.

```
          FILE_IDENTIFIER
          ________________________________________
     BODY:
          NODE_TYPE
          BASE_IDENTIFIER
          ADJACENT LOCK NODES – NDLL
          APPLICATION DATA
     EDGE:
          NODE_TYPE_1
          BASE_IDENTIFIER_1
          NODE_TYPE_2
          BASE_IDENTIFER_2
          APPLICATION DATA
     EDGE:

        ⊕
        ⊕
        ⊕

          ________________________________________
```

Fig. 6—Generic node structure.

9.2.1 The node-type

In this application node-type refers to the type of object the node is modeling and is one of: cable, pair, terminal, loop, ... GRIT uses it to identify the file and program formats for performing the conversion from file to program records and vice versa. It is also used for debugging and by the database auditing program.

9.2.2 Base-id

Because of the requirement to retain the state of the inventory before and after any future assignment of facilities, several versions of a node may have to be retained. One represents the actual state of the world and the others represent future possible states. For example:

(*i*) Loop LP is serving customer AJG at living unit LU at this time;
(*ii*) AJG is moving out of LU in two weeks;
(*iii*) Customer ZN will be moving into LU in three weeks;

The assignment programs receive versions of nodes showing the state of the inventory after AJG moves out and see that loop LP at LU is available; LP is assigned to serve ZN at LU.

In this case, there are three versions of loop LP: one for now, one for when AJG moves out, and one for when ZN moves in. Each version is a node and each has the same base-id in its body. But, each has a different file-id with the "now" version's file-id and base-id being identical. Any edge that points to these nodes does so ambiguously because it uses the base-id. The pending routines assure that the version given to any application routine is consistent with the inventory state the routine is working in. The base-id allows the pending routines to be transparent to the application programs. (See the supplement on pending Section X.)

9.2.3 The lock nodes

The lock nodes make up the node lock list, the NDLL, that defines the directed acyclic node graph for locking, the DANGL. See the concurrency and locking section (Section XI.)

9.2.4 The application data

The application data is that information needed to model the loop plant aspects of the object that the node represents. Typically there is little data here. Most of the node's meaning is in its relationships to other nodes, the edges. In the pair 121 cable 101 example, the data are the remark (for human, not computer consumption) and the fact, (F1, no), that the pair does not originate at the central office.

9.3 Edge

Each edge may point to one node or to two nodes. Though in principle an edge could point to any number of nodes, in this application at most two are ever needed.

9.3.1 The facility type

This datum allows one to know what kind of object(s) the edge points to without the cost of fetching the node(s) from the file. The redundancy buys performance. The audit program uses the facility type to validate the topology of the database's hypergraph.

9.3.2 The base identifier

The base identifier has all the properties and functions discussed under the body's base-id. In particular, the invariance of the base-id under external-name changes is crucial to the stability, maintainability, and reliability of the database.

9.3.3 The application data

These data are the most important application data in the node. The relationships of one object to another are the meat of a database. The properties of an object (the body's application data) are isolated, relatively unimportant facts. Whether a terminal is of type cross-connect or distribution and how big it is is not nearly so important as what pairs appear in it and which pairs are connected to which.

X. PENDING—SUPPLEMENT

The purpose of pending is to provide an interface between the database and the application programs so that those programs can be written as if pending did not exist. The goal is attained because of two major aspects of the pending logic:

(*i*) First, every request to read an object is intercepted by pending, which returns the proper pending version of the object.

(*ii*) Second, pending calls the application programs, rather than the reverse. The call is made by a different pending driver routine for each transaction type (inquiry, inventory modification, loop construction, etc.). The drivers are usually quite small and perform the following tasks:

a. they initialize pending structures;
b. they invoke an application program (possibly iteratively);
c. before returning, they delete pending structure debris.

The action of pending will be explained by a detailed examination of the following example. The customer at the living unit of the example (Fig. 2) notifies the telephone company of plans to move out in two weeks. An order to disconnect service is entered (via a transaction into the computer system) with a due date of "now + 2 weeks." When the move takes place, a "completion" transaction is entered. In between, the database must show the state of the inventory before the disconnect order and the state after the completion.

In the before state the loop at the living unit has the status "working" and its external name is, say, 999-6666. All that really must be done to change the before state to the after state is to change the status to "idle" and to remove the relation between the loop node and the name node for telephone number 999-6666.

The left side of Fig. 7a shows the before state. There are edges relating the loop node to: the two pair nodes, the living unit node, and the name node for 999-6666. Figure 7b shows the after state with the status changed to "idle" and the relation to the name node deleted. In Fig. 7, data irrelevant to the discussion are suppressed.

10.1 Processing the disconnect order

The simple sentence, "All that really must be done is . . . ," expands into a not-so-simple scenario. First recall that each transaction type has a driver that uses the pending routines and invokes application routines that are (almost always) ignorant of pending. In this case the driver is PDD, the pending driver for disconnect orders. Now to the scenario, which must be read together with Figure 7a.

(*i*) PDD creates the pending node, PN, that will tie together the nodes that constitute the pending state created by the disconnect order. PDD calls the disconnect program, which is written as though pending did not exist.

(*ii*) The disconnect program invokes a primitive (with the telephone number as an argument) that should fetch loop node 232 in

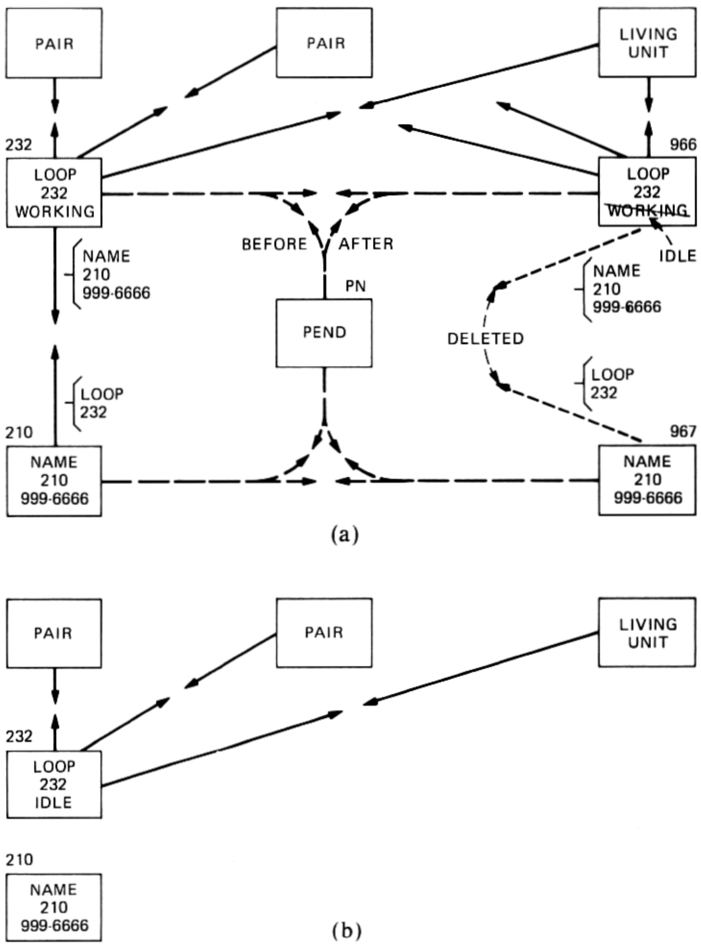

Fig. 7—(a) Pending disconnect order. (b) Order completed.

order to update its status and telephone number association. (This primitive's actions were described in the section on primitives.)

(*iii*) One of the primitive's actions is to call HDBMS to fetch loop node 232 for update. That call is intercepted by pending. First, pending manufactures a clone of loop node 232 by creating a new node, node 966, and copying loop node 232 into it; second, pending relates loop node 232, its clone 966, and PN; third, pending writes the updated node 232 to the file; and finally, they return loop clone 966 to the disconnect program instead of loop node 232, which was requested!

(*iv*) (The disconnect program could peek behind the curtain, i.e., look at the file-id, and discover the fraud, but it was designed to have no need to know and to live in blissful ignorance of pending.) The program changes the status from "working" to "idle" and calls a primitive to remove the loop's name.

(*v*) The primitive searches the edges of clone 966 for one whose type is "name". That edge points to node 210 and the primitive calls HDBMS to fetch it for update.

(*vi*) The actions of Step (*iii*) are repeated and again the call is intercepted by the pending routines. First, pending manufactures a clone of name node 210 by creating node 967 and copying node 210 into it; second, pending relates nodes 210, 967, and PN; third, pending writes the updated node 210 to the file; and, finally, pending returns the clone 967 to the primitive.

(*vii*) The primitive now thinks it has loop node 232 and name node 210 and is deleting the edge in 232 pointing to 210 and vice versa. Actually it has the loop clone 966 and name clone 967 and it deletes the edge in 967 pointing to name node 210 and the edge in 967 pointing to loop node 232. It writes the updated node 967 to the file (thinking it is 210) and returns to its caller, the disconnect program.

(*viii*) The disconnect program has done its job of changing the status of the loop to "idle" and deleting the telephone number. It writes loop node 966 to the file (thinking it is 232) and returns to its caller, the driver PDD.

(*ix*) The PDD writes node PN to the file and thus finishes the disconnect transaction.

PN has hyperedges relating the before and the after nodes so that inquiries about the loop can produce all states of the inventory.

10.2 Processing the completion order

Two weeks later, the customer moves out and a transaction is entered to complete the removal of telephone service. The following actions by PDC, the pending driver for order completion, convert Fig. 8a to 8b:

(*i*) The contents of each before node are replaced by the contents of its corresponding after node (with its hyperedge to PN deleted). In this example, replace the contents of 232 and 210 with those of 966 and 967 (with their hyperedges to PN deleted).

(*ii*) Write the updated before nodes to the file.

(*iii*) Delete the after nodes.

(*iv*) Delete node PN.

Note: PDC has no knowledge of the application program!

10.3 Implementation of pending

The implementation of pending implied by the example uses the hypergraph database structure directly, first, by its use of the PN node and second, by its use of a modified copy of a node to represent a future state of an object. This is only one way to implement pending.

In Ref. 4, an abstract model of pending identifies three important pictures of the database that completely characterize a pending database.

One picture shows the history of a single object. For example, the picture containing nodes 232 and 966 shows the history of the loop. A second picture shows the changed nodes produced by a transaction. For example the picture containing loop node 966 and name node 967 shows the effect of the disconnect order. The third picture gives a consistent cross-section of the database that contains, for each object, one node representing its state. (One can think of this as a snapshot at a point in time.) For example, the snapshot of the state of affairs as it looked before the disconnect order consists of the two pair nodes, the living unit node, loop node 232, and name node 210. Similarly, the snapshot of the state of affairs after the disconnect order is completed consists of the previous snapshot with nodes 232 and 210 replaced by 966 and 967.

These three views of the pending database clarify the complex role played by the PN node and enable one to identify and evaluate other kinds of implementations.

In the present implementation, each state of an object is represented by a separate node. If the change from one state to a successor state requires a small change to a node's data, then the representation wastes space. A representation that stores only the data change conserves space, but complicates reading a node. For applications that have very large objects with simple structures (e.g. an array), both representations may be necessary.

XI. CONCURRENCY AND LOCKING

In this section the term node is used ambiguously to mean node in the sense used in previous sections or in the more general sense of a record. The context will distinguish between the two.

The local loop plant is partitioned into units called wire centers. Each consists of a central office and living units, pairs, etc. associated with it. The size of a wire center's database is approximately proportional to the number of pairs in it and may have from 0.1 to 5 million nodes. The number of transactions that must be processed per wire center per day is approximately proportional to the number of working loops. In medium and large wire centers, transactions must process concurrently to achieve the necessary throughput. Fortunately, a concurrence of 2 or 3 suffices.

Each time a locking mechanism is invoked, it locks a set of nodes called a locked-set and the average size of the locked-sets is called the granularity of the locking mechanism. The finest granularity locks one node at a time. The coursest granularity locks all nodes at a time. Each locked-set has a lock-node associated with it that the file system locks. The file system also notes the identity of the invoking transaction. Any transaction that wishes to access a node in a locked-set must have permission to access its lock-node. The locking of a node by the file system takes time and space. This section proposes a locking mechanism that balances locking costs and concurrency requirements.

The large number of nodes that some transactions must examine prohibits the use of a fine granularity. However, the low concurrency requirement of 2 or 3 permits the use of a coarse granularity as long as the locked-sets of concurrent transactions intersect only when the transactions examine the same part of the loop plant.

Let us examine three kinds of locked-sets: fixed, hierarchical, and dynamic hierarchical. For fixed locked-sets, all the nodes are divided into disjoint sets and the set to which a node belongs never changes. The loop plant of a wire center has a complexity that is only hinted at in Fig. 1. Moreover, the addition and rearrangement of cables, pairs, and terminals causes the topology to change. We have not been able to design a simple locking mechanism based on fixed locked-sets.

Hierarchical locked-sets can be defined in databases in which each node can be reached by a unique "access path" through the nodes of the database. The path's origin node, a lock-node, is locked by the file system before fetching the desired node. Hierarchical locking mechanisms require that each node's lock-node be known at compile time. In our database, a node can be reached from a multitude of directions, either via the name table or via any of the node's adjacent nodes. Moreover, none of the access paths are known until execution time. Hierarchical locked-sets cannot be implemented in this database.

The idea of dynamic hierarchical locked-sets is derived from looking at hierarchical locked-sets from a different point of view. Suppose one reverses the direction of an access path so that it starts at the node to be read and ends at its lock-node. The collection of directed edges of reversed access paths defines a forest of trees rooted at the lock-nodes. We call the forest a lock graph. Now, rather than accessing a node through its access path, the lock graph is traversed from the node to its root in order to identify the node's lock-node. This different view of hierarchical locked-sets can be generalized so that the lock graph is a directed acyclic graph (DAG). Moreover, the DAG can change with time as loops, pairs, terminals, etc. are added and deleted from the database. It can also incorporate pending nodes. In hierarchical locking, a node's lock-node is known at compile time, while in dynamic locking the lock-node is found by traversing the lock graph.

The dynamic hierarchical locking mechanism was selected for this application. The lock graph is called a DANGL—a Directed Acyclic Node Graph for Locking. It is a spanning subgraph of the hypergraph of the database, that is, every node is in it. A node is locked by walking from it to each of its lock-nodes and then calling the file system to lock them.

Figure 8 shows the portion of the DANGL associated with the example. Each node has one or more edges pointing to adjacent nodes that are closer to the central office. In this portion of the DANGL, all the lock-nodes are F1 cables. The DANGL is defined by the following set of rules that are special to the loop plant. The possibility of using the DANGL technique in another application critically depends upon being able to find similar rules.

(*i*) A living unit has edges to its working and serving terminals.

(*ii*) A terminal has edges to all cables that appear on its central office side.

(*iii*) A loop has an edge to its living unit.

(*iv*) A pair has an edge to its cable.

(*v*) A cable has an edge to the terminal on its central office side. (The loop plant requirements allow only one such terminal.)

(*vi*) All other node types are isolated lock-nodes.

11.1 Locking algorithm

The locking algorithm is based on the following.

11.1.1 The DANGL

The DANGL is a Directed Acyclic Node Graph for Locking. It is a subgraph of the hypergraph of the database and is constructed so that it contains every node in that database. The DANGL is maintained by

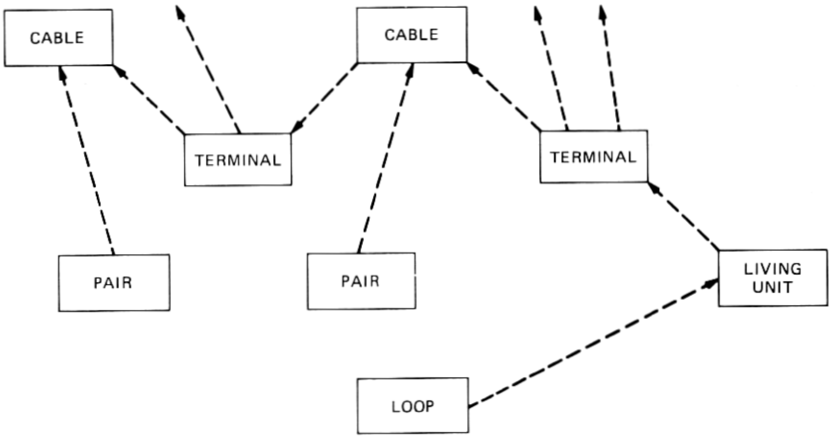

Fig. 8—The DANGL—the portion associated with the example.

inventory programs and is transparent to all other application programs. It is dynamic and reflects plant changes and pending. The DANGL is broad—it contains all of the nodes in the database. It is shallow—almost every node is 2 or 3 edges from its lock-nodes. And it is clannish—there are few lock-nodes (less than 50).

11.1.2 The NDLL

The NDLL, the Node Lock List, is a list of a node's adjacent neighbors in the direction of the lock-nodes. It represents the DANGL and is defined by the above rules. The NDLL of a lock-node is empty.

11.1.3 The TRATLL

The TRATLL, the inTRA-transaction Lock List, serves to decrease the cost of locking. Each transaction has a TRATLL that is maintained by the HDBMS. The TRATLL is a list of all the nodes in the short paths from a node to its lock-nodes. Before traversing the DANGL to find the lock-nodes of a node, the TRATLL is searched to see if the node is already locked.

11.1.4 The FLOG

The FLOG is the mechanism for File system LOckinG a node's file record. FLOGing is expensive and is performed by the computer's file system, which also handles deadlock detection, deadlock resolution, rollback, and recovery. The FLOG locks the few lock-nodes of a node. Most transactions need to lock the lock-nodes of only one node in order to protect the portion of the database that it will use.

11.1.5 Cheap_lock

Cheap_lock is a routine that is called by HGDMS to lock a node's lock-nodes.

```
procedure cheap_lock (fid)              /* node's file id */
   if fid is in TRATLL then return;
   node_a = read (fid);
   if node_a's NDLL is empty then       /* fid is a lock-node */
      FLOG the fid;
      add fid to TRATLL;
      return;
   else                                 /* fid is not a lock-node */
      cheap_lock (node_a's NDLL);       /* lock every id in NDLL */
/* has fid's NDLL changed while we were locking its ancestors? */

   node_b = read (fid);
```

```
  if node_a's NDLL                    /* no change */
equals node_b's NDLL then
     add fid to TRATLL;
  else                                /* NDLL has changed!! */
     cheap_lock (fid);                /* start all over */
  return;
```

11.2 Why it works

The path to a lock-node is always short because the DANGL is shallow. There are only a few short paths from a node to all of its lock-nodes because the DANGL has few lock-nodes and most nodes have few outward directed edges.

Let us invert our point of view to look back from L(N), the set of lock-nodes of a node N. From L(N) we can see the fanout set of N, F(N), containing all nodes that have at least one lock-node in L(N). F(N) is large because the DANGL is broad and has few lock-nodes and thus the locking mechanism will work only where low concurrency is required. A little thought shows that F(N) contains almost everything that is related to N in the hypergraph. Thus in locking N, a transaction insures that almost no other transaction can touch N's relatives. This practically eliminates deadlock.

11.3 What routines know about concurrency

The routines that add new objects to inventory must initialize each new node's NDLL with its adjacent nodes in the DANGL. The HDBMS uses the NDLL to walk the DANGL. The HDBMS also maintains the TRATLL to streamline locking. The file system FLOGs the lock-nodes. Locking is transparent to all other routines including pending.

XII. ACKNOWLEDGMENTS

Many people have contributed to this project: the original model of the local telephone plant as a graph and initial database design—Richard S. Ferrer, A. Jay Goldstein, Marion S. Newton; additional modeling, generalization to a hypergraph—A. Jay Goldstein, Zaher Nazif; pending—Gerald C. Boyle; inventory logic—Zaher Nazif; local loop assignment logic—Joseph R. Dudis; locking and concurrency—A. Jay Goldstein, Terry R. Nusser; hypergraph database management—Robert D. Harrison; guru and GRIT inventor—James E. Weythman; managerial and moral support—Larry Bernstein, D. Lou Casulli, Terry L. Warner, and Barbara T. Bauer. M. Douglas McIlroy's thorough critique of a draft of this paper is deeply appreciated.

REFERENCES

1. P. Chen, *The Entity-Relationship Model—Towards a Unified View of Data*, ACM Transactions on Data Base Systems, *1,* No. 1 (1976), pp. 9–36.
2. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Englewood Cliffs, New Jersey: Prentice Hall, 1978.
3. C. Berge, *Graphs and Hypergraphs*, New York: North-Holland, 1973.
4. G. C. Boyle, N. Dowuona, and A. J. Goldstein, *An Abstract Model of Pending*, unpublished work.